THESIS

JING LIN 2011

CLASSIFICATION OF ABELIAN SQUARE-FREE WORDS OVER THREE LETTERS

Rovaniemen ammattikorkeakoulu
University of Applied Sciences
LUC

DEGREE PROGRAMME IN INFORMATION TECHNOLOGY

Rovaniemen
ammattikorkeakoulu
University of Applied Sciences

ROVANIEMI UNIVERSITY OF APPLIED SCIENCES
SCHOOL OF TECHNOLOGY

Degree Programme in Information Technology

Thesis

CLASSIFICATION OF ABELIAN SQUARE-FREE WORDS OVER THREE LETTERS

Jing Lin

2011

Supervisor: Veikko Keränen

Approved _____2011_____

The thesis can be borrowed.

Rovaniemen ammattikorkeakoulu
University of Applied Sciences
LUC

School of Technology
Degree Programme
In Information
Technology

Abstract of Thesis

Author Jing Lin

Year 2011

Subject of thesis Classification of Abelian Square-Free Words over Three Letters

Number of pages 32 + 2

The goal of the thesis was to find the method to construct abelian square-free words over three letters based on the knowledge of the abelian square avoidance. Four programs were created to accomplish the goal of the thesis by using the IDLE integrated development environment and the Python programming language.

The programs and their input files were uploaded to computers, Hippu and Louhi, on the IT centre for science (CSC) by using the Nutty SSH client program which supports the means for security communication. Two programs were successfully run on Hippu which is meant for running interactive jobs and background jobs. Another two programs failed to run on Louhi which is meant for running parallel jobs and batch jobs.

Many versions of both the programs were tested on Hippu. Around a million of the abelian square-free words of the different lengths over three letters were constructed on Hippu during the thesis project. The programs which are attached in the appendices were not developed as fast as expected.

The most difficult problem was to build a parallel program in Python and to find the way for it to run on Louhi. Many tests were done on Louhi, but all of them failed. For future research, the programs' running speed could be improved significantly, if a solution to these problems can be found.

Key words abelian square avoidance, combinatorics on words, Python, supercomputer

CONTENTS

LIST OF FIGURES

LIST OF TERMS AND SYMBOLS

Alphabet — An *alphabet* $\sum$ is a finite non-empty set of abstract symbols called letters.

Word — A *word* over the fixed alphabet $\sum$ is a finite string, or sequence, of letters belong to $\sum$.

$|w|$ — The *length of the word* w, denoted by $|w|$, is the number of occurrences of letters in w.

λ — The *empty word*, denoted by λ, is the neutral element of catenation.

$|w|_x$ — The *number of occurrences of one letter* $x \in \sum$ in w.

$\psi_{\sum}(w)$ — The *Parikh vector* of w, i.e., $\psi_{\sum}(w) = (|w|_1, \cdots, |w|_n)$. Usually the subscript $\sum$ is omitted and written simply ψ instead of $\psi_{\sum}$.

Catenation — For words u and v the juxtaposition uv is called the *catenation* (or concatenation) of u and v.

Factor — A word u is called a *factor* of a word w, if $w = pus$ for some words p and s.

$pref(w)$ — The set of all prefixes of a word w.

$pref_k(w)$ — The prefix of length k of a word w.

$\sum^*$ — The set of all finite words over $\sum$.

$\sum^+$ — The set of all finite nonempty words over $\sum$.

$\sum^\omega$ — The set of all infinite words over $\sum$.

Monoid — Let G be a non-empty set and let $*$ be a binary operation defined on this set. A pair $< G, * >$ is a *monoid*, if the following properties hold:

- $a * b \in G$ whenever a and $b \in G$ (G is closed with respect to the binary operation $*$)
- $(a * b) * c = a * (b * c)$ for all $a, b, c \in G$ (associativity)

- there exists a *neutral* element e such that $e * a = a * e = a$ for all $a \in G$.

In combinatorics on words, the binary operation $*$ is just the catenation of words.

Free monoid

A *monoid* G is called *free,* if it has a subset B such that each element of G can be uniquely expressed as a product of elements of B.

Morphism A *morphism* h is a mapping between free monoids $\sum^*$ and Δ^* with $h(uv) = h(u)h(v)$ for every u and v in $\sum^*$. Thus a morphism is compatible with the catenation of words. Consequently, a morphism is uniquely defined, if the word $h(x)$ in Δ^* is given for each letter x in $\sum$.

1 INTRODUCTION

Combinatorics on words is a popular topic which has been discussed and researched for many years. It can be applied, for example, to number theory, algorithmic music, cryptography, compression of data, and bioinformatics. The study of the abelian square avoidance belongs to the field of combinatorics on words.

The thesis is about the theory of the abelian square avoidance and the practical approach to construct abelian square-free words over three letters by using computers. The thesis discusses the restricted abelian square avoidance which allows the repetitions of length = 1.

The thesis is divided into three parts. The first part introduces the basic knowledge of combinatorics on words and abelian square avoidance. The second part explains the program which was created to construct abelian square-free words over three letters (here xx is allowed for a letter x). The third part describes the runtime environment of the program and the steps needed for running the program.

Chapter 2 gives definitions and explanations in the study of word structure, pattern avoidance, abelian square avoidance, and related fields. Chapter 3 explains two programs which construct the so-far-favourable abelian square-free words in different ways. Chapter 4 describes the background and computers of the IT centre for science (CSC). Chapter 5 explains the process for running the program on the CSC computer.

2 PRELIMINARIES

2.1 Combinatorics on Words

Combinatorics on words is a study of word structures. Words, in turn, are basic objects of formal languages. The field of combinatorics on words was initiated by Axel Thue in 1906 (Keränen 2009a, 3893). In mathematics, a (formal) language is a set of words. A word over $\sum$ is a finite string, or sequence, of letters that belong to $\sum$ (Keränen 2009a, 3895). For example, in the case of $\sum = \{a, b, c\}$, a word $w = bacaabc$ and a formal language $L = \{abccba, ba, caa, bacaabc, aabb\}$ can be written. In this example, the word $w = bacaabc$ belongs to the language L ($w \in L$) and the length of this word w is six ($|w| = 6$). According to the same example, it can be said that $|w|_a = 3$, which means that the number of the occurrences of the letter a ($a \in \sum$) in the word w is three. Moreover, the Parikh vector of w can be represented by $\psi(w) = (3, 2, 2)$.

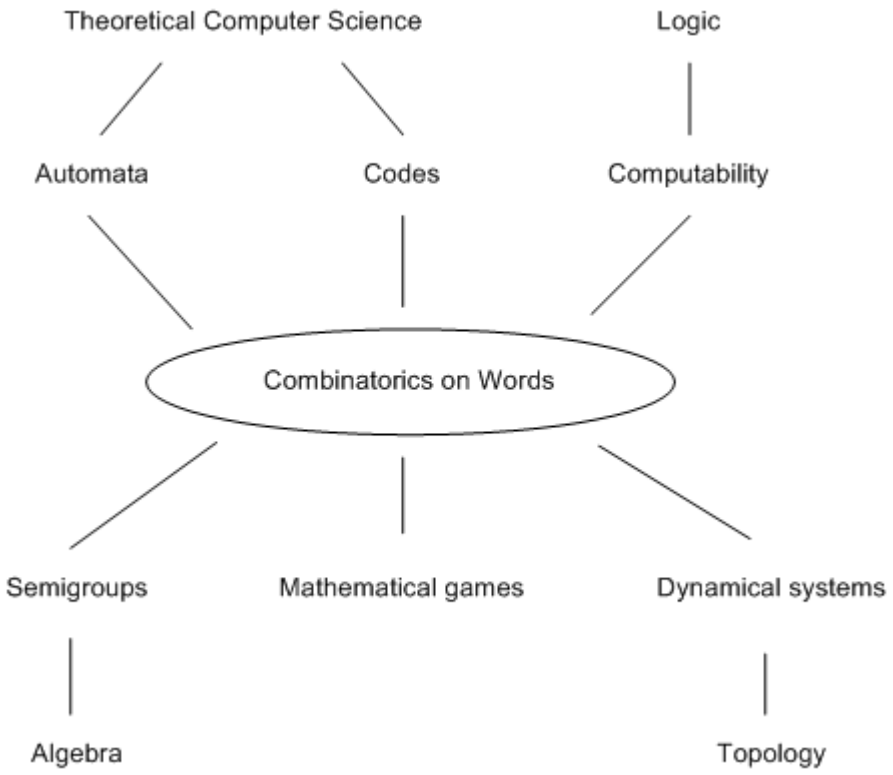


Figure 1. Connections between Combinatorics on Words and Other Fields (Karhumäki 2011)

Combinatorics on words is a part of combinatorics and number theory (integer sequences). It has become a separate subject of its own. Combinatorics on words connects to many other fields, for example, algebraic semi-groups and free monoids. More connections can be found in Figure 1 above.

2.2 Ordinary and Abelian Pattern Avoidance

The study of the abelian square free words is a part of combinatorics on words. More details can be found in Figure 2 about the relationship between combinatorics on words and abelian square-free words.

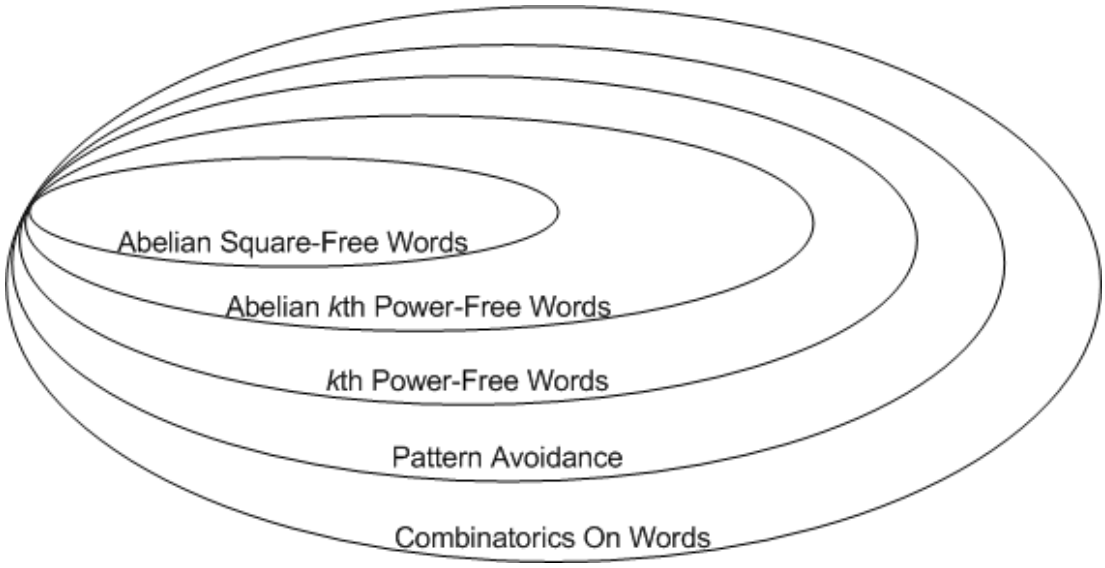


Figure 2. Relationship between Combinatorics on Words and Abelian Square-Free Words

In the study of word structures, i.e., combinatorics on words, there is a field called pattern avoidance which was, in a sense, introduced by Bean, Ehrenfeucht and McNulty (1979). Let X be a set of unknowns (disjoint letters), and $\sum$ a set which is the 'usual' alphabet. A *pattern* is any nonempty word p in X^+. It can be said that a *pattern* is *avoidable* in $\sum$, if there exists an infinite word $\omega \in \sum^{\omega}$ such that, for any morphism $h : X^+ \to \sum^+$, the word $h(p)$ is not a factor in ω. If pattern p is not avoidable in $\sum$, then it can be said that, pattern p is unavoidable in $\sum$ (Karhumäki 2011, 48). This is the definition of pattern avoidance in ordinary (non-abelian) case. For instance, let $\sum = \{a\}$ and pattern $p = yy$ in X^+, where X is the set of unknowns.

Consider a morphism $h : X^+ \to \sum^+$, $h : y \to aa$. Any infinite word ω is of the form $aaaa\cdots$, and therefore contains, as a factor an unfavourable pattern $h(p) = h(yy) = aaaa$. In this case, it can be said that, the pattern p is unavoidable in $\sum$.

Basically, one can define many kinds of different patterns and study their avoidance properties. The most common patterns are k-repetitions, i.e., *k*th powers. In case of 2-repetitions and 3-repetions, the patterns are called as squares, cubes respectively. A *square* is a word of the form uu with $u \neq \lambda$. A *cube* is a word of the form uuu with $u \neq \lambda$ (Karhumäki 2011, 33).

In generally, a *word* ω is a *kth power,* if $\omega = u^k = pref_{k\cdot|u|}(u^\omega)$ with $u \neq \lambda$ (Karhumäki 2011, 33). The *kth power-free* words are the words which do not have the pattern $p = \underbrace{uu\cdots u}_{k}$ as a factor.

Abelian kth power-free words do not have the pattern $p = \underbrace{p_1 p_2 \cdots p_k}_{k}$ as a factor, where all words p_i are permutations (anagrams) of each other (Keränen 2009b, 358).

2.3 Abelian Square Avoidance

A word is called *abelian square-free*, if it does not contain any abelian square as a factor (Keränen 2009a, 3894). An *abelian square* is a nonempty word uv, where u and v are permutations (anagrams) of each other (Keränen 2009b, 358). Specially, for researching abelian square-free words over a three letter alphabet, one can investigate what happens when the repetitions xx and xxx for letter x are allowed, that is, here repetitions of length = 1 would be allowed. As it was mentioned in the chapter 1, in this thesis only the restricted abelian-square avoidance is discussed.

Let $\sum = \{a, b, c\}$. In this case, the words aa, aaa, bb, bbb, cc, and ccc would be allowed, but no longer abelian squares will be accepted. For example, the words $\underbrace{abc}_{u}\underbrace{cba}_{v}$ and $\underbrace{accbac}_{u}\underbrace{abccca}_{v}$ are abelian squares that should be avoided in this three letter case.

In Figure 3, there are three occurrences of factors u and v in the word w, for which $|u| = |v|$. However, in this thesis, the adjacent case is only

studied. In here, it is called catenation, i.e., juxtaposition uv, factors u and v are called catenated factors (Keränen 1986, 7-8).

The permutation of the word u can be thought as a word v, where the occurrences of each letter is the same as in u, that is, u and v are anagrams of each other. In here, it is considered an example of an abelian square-free (allow xx) word $w = aacabcb \in \sum^* = \{a, b, c\}^*$. Indeed, there are no catenated factors of length > 1 which are permutations of each other. So w is an abelian square-free (allow xx) word. Another example of a word that is not abelian square-free is as follows. Let $\sum = \{a, b, c\}$, and consider the word $w = abbacaa$. The factors ab and ba are permutations of each other of length > 1, so the word w is not abelian square-free.

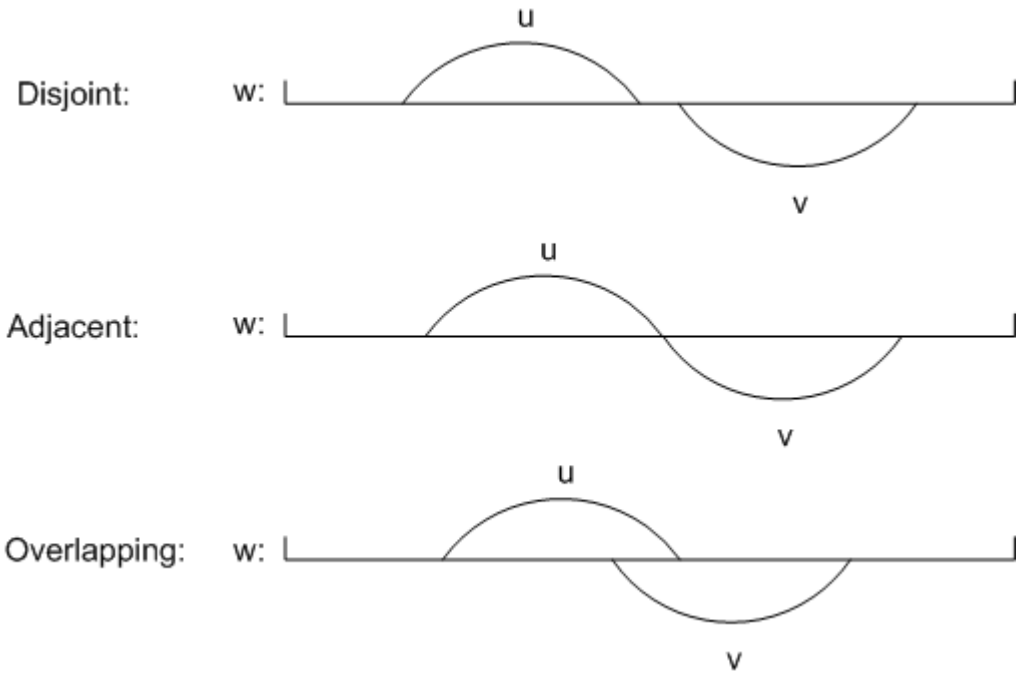


Figure 3. Three Occurrences of Factors in a Word (Karhumäki 2011, 34)

For a longer word, for example, $w = \underbrace{aaabaaacaaab}_{u}\underbrace{bbaaaccbaaca}_{v}$, there is a good way to use Parikh vector as a presentation tool for checking, whether factors u and v are permutations of each other. In the word w, factor u and v can be presented by Parikh vectors as $\psi(u) = (9, 2, 1)$, $\psi(v) = (6, 3, 3)$. Thus $\psi(u) \neq \psi(v)$. Therefore, factors u and v are not permutations of each other, and the word w is abelian square-free (allow xx).

2.4 Classification of Abelian Square-Free Words over Three Letters

According to Keränen (Keränen 2009a, 3895), the abelian square-free words over three letters are classified into three categories, favourable (good) words, so-far-favourable (so-far-so-good) words, and unfavourable (bad) words.

The words are classified according to the situation of the extension of themselves. The word is extended by adding one letter which belongs to the alphabet $\sum$, to its left side (or right side) at a time. If the word cannot be extended to the target length due to the occurrence of the abelian squares as factors, in this case, it is called a bad word. Otherwise, the word is called so-far-so-good word. The difference between so-far-so-good word and good word is that, the former is just a finite word, and the latter can be extended to an infinite word. Good words are from the list of the so-far-so-good words and can be proved to be extendable in an abelian square-free style to infinity. The program is designed by following this classification to find so-far-so-good words that are as long as possible. Later, perhaps some experts can figure out, whether these so-far-so-good words can be proved as good words. Some examples and the more details of the word extension are given in chapter 3.

3 PROGRAMS FOR PRACTICAL WORK

3.1 General Goal of Programs

The general goal of the programs is to help research, whether abelian squares (allow xx) can be avoided over three letters. For this purpose, the programs focus on constructing words and then checking, if the constructed words are abelian square-free over three letters. The main idea of the program can be divided into four parts to achieve the purpose.

The first part is to read the base words from the input file. The second part is to extend the base word by adding one letter at a time. The third part is to check, if the extended word is abelian square-free. The forth part is to output the result, the so-far-so-good word or the bad word, each to a respective file.

3.2 Program for Preparation

3.2.1 Purpose of Preparation Program

The preparation program named ‘preparation_asf_3L_v1.0.py’ (see Appendix 1) can build the lists of all the so-far-so-good words and bad words length by length. However, for constructing the longer lengths of the abelian square-free words, for example, length > 24, it will consume a very large amount of time depending on the computer capacity and the range of lengths the user wants to construct. For this reason, preparation_asf_3L_v1.0.py is used to build the lists of the so-far-so-good words and bad words only in the range of length = 5 to length = 24. Then the final output of the preparation program, the text file contains the list of all the so-far-so-good words in length = 24, is one of the inputs for the further procedure, which will be introduced in chapter 3.3.

3.2.2 Overview of Preparation Program

In the preparation program, there is main function and fourteen other functions. Two functions check and load the text file which gives the base words as input. One function that extends words, will be introduced in chapter 3.2.3. Three functions check, if the extended words are abelian square-free. Four functions control the format of the table in the text file which is used to record runtime. One function counts the processing time for

constructing all the so-far-so-good words and all the bad words for each length. The rest three functions print the lists of the bad words and so-far-so-good words to respective files.

3.2.3 Explanation of Preparation Program

Two tasks form the core of the preparation program. The first task is to extend the base word by adding one letter as its prefix or suffix depending on the length of the base word is odd or even (see Figure 4).

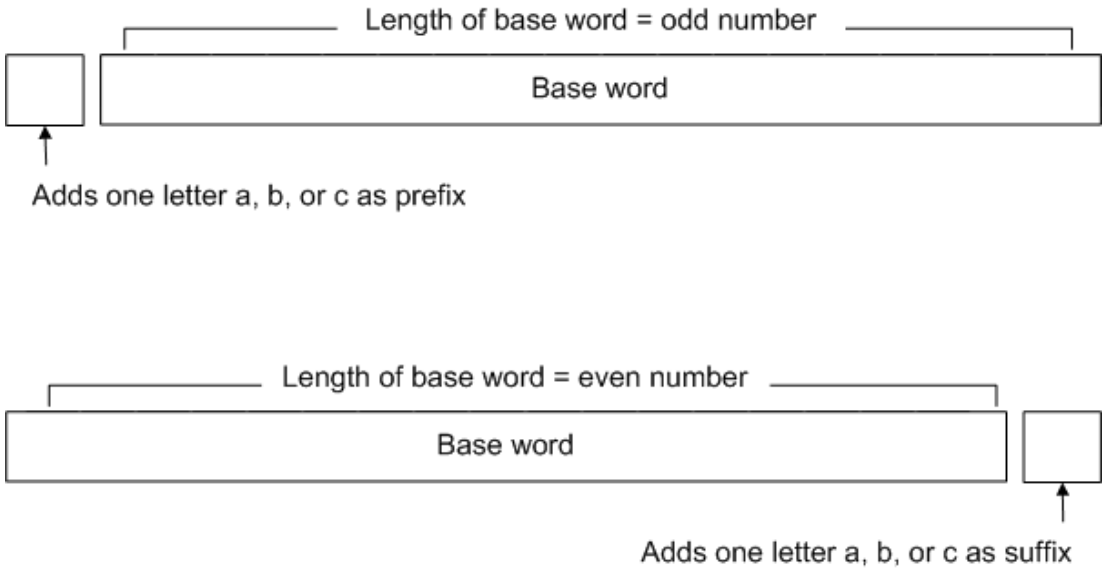


Figure 4. Process of the Function for Extending Word

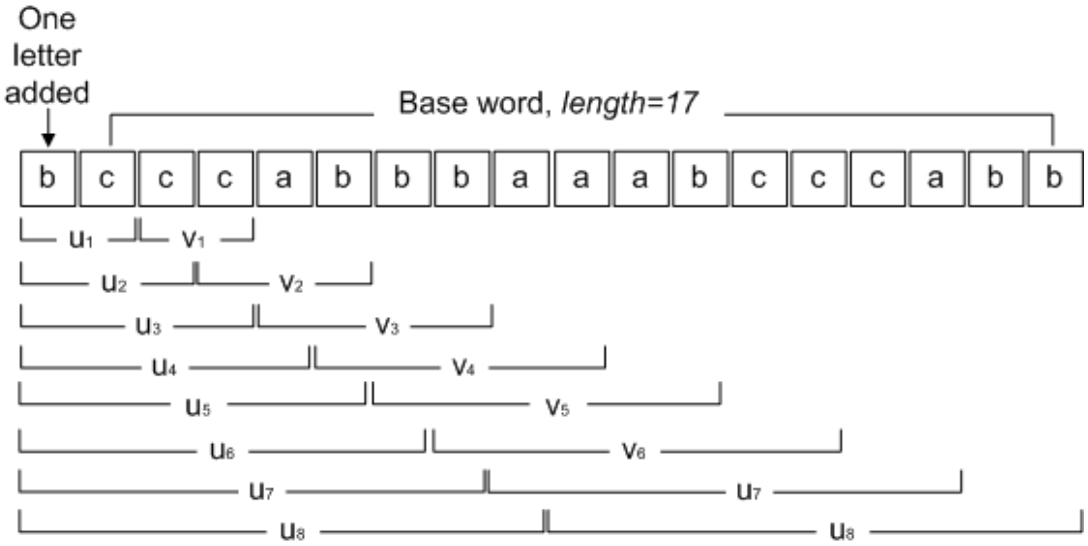


Figure 5. Example of the Process in a2fCheckL Function (length = 18)

The second task is to check, if the extended word is abelian square-free. There are two functions for checking the word. The function named 'a2fCheckL' checks the extended word, the length of which is even, from its left side (see Figure 5). The function named 'a2fCheckR' does the other way around for the extended word, the length of which is odd (see Figure 6).

Figure 5 and Figure 6 show the sum of the length of the factors $|u_i|$ and $|v_i|$ is even. Indeed, because of $|u_i| = |v_i|$, and then $|u_i| + |v_i| = 2|u_i| = 2|v_i|$. This means for the odd length of the extended words, one letter on most left side is always omitted by the checking function (see Figure 6). However, it is not a problem, because the first letter from left side has been checked in its base word $cccabbbaaabcccabbb$ already (see Figure 5).

During the checking loops, the function a2fCheckL or a2fCheckR first collects factors u_1 and v_1 from the extended word. Then the function checks, if $\psi(u_1) = \psi(v_1)$, i.e., if u_1 and v_1 are permutations of each other. If not, the function continues to do the same process to u_2 and v_2. If the function finds $\psi(u_i) = \psi(v_i)$, the checking loops will be ended, and the extended word will be considered as a bad word and saved in the list of the bad words. Otherwise, if for all the factors $\psi(u_i) \neq \psi(v_i)$, the extended word will be considered as a so-far-so-good word and save in the list of the so-far-so-good words. Until all the base words from the input file are extended and checked, the program prints the list of the bad words and the list of the so-far-so-good words to respective files.

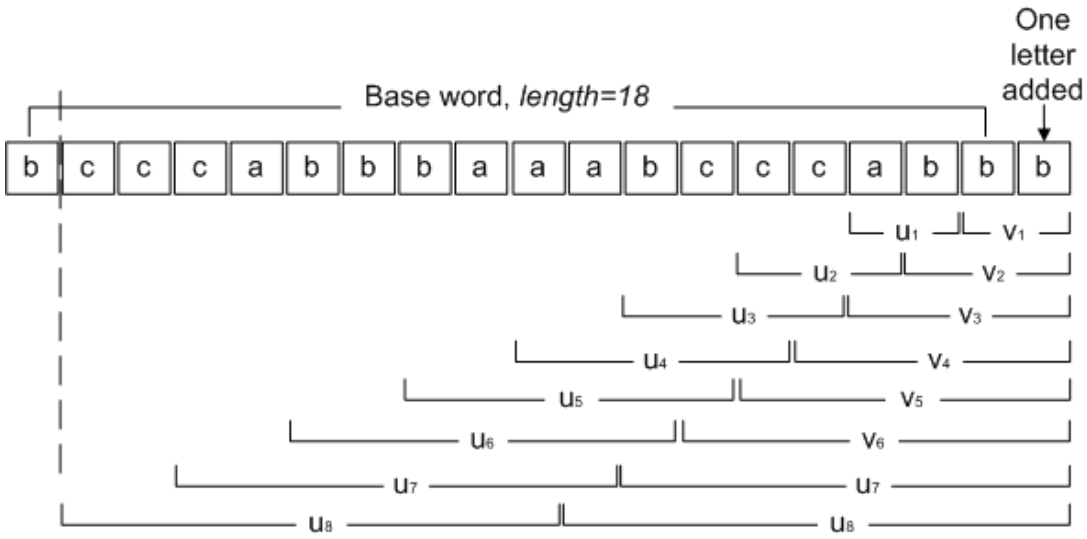


Figure 6. Example of the Process in a2fCheckR Function (length = 19)

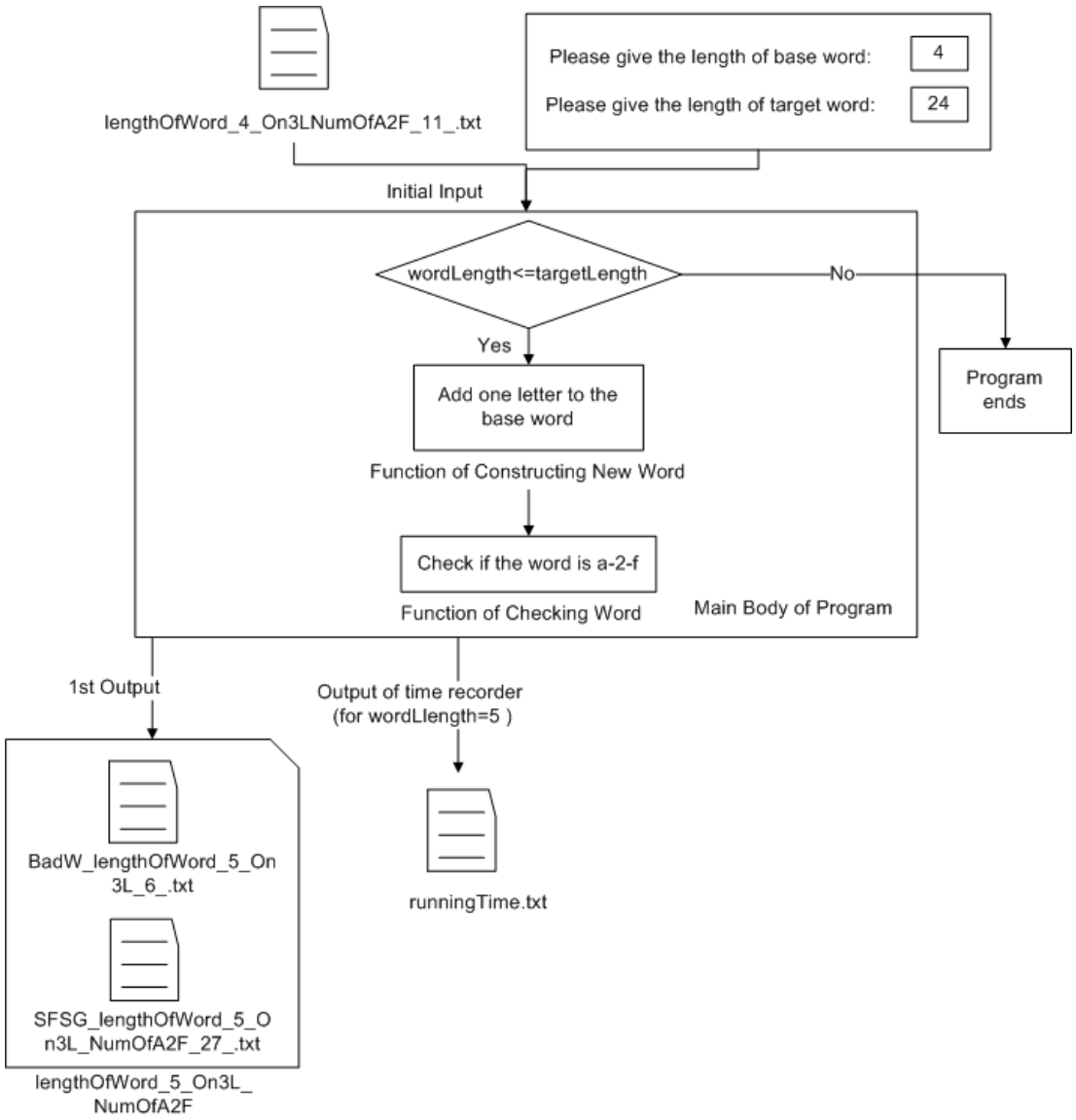


Figure 7. First Stage of the Loop in Preparation Program

In the preparation program, there is a loop used to control the program doing the tasks until the condition of the loop is not satisfied. There are three items as the initial inputs for the program. The range of the word length, i.e., start point and end point, are used in the loop condition to control the program. A text file contains eleven so-far-so-good words in length = 4 ($aaab$, $aaba$, $aabb$, $aabc$, $abaa$, $abac$, $abbb$, $abbc$, $abca$, $abcb$, $abcc$), from which all the other so-far-so-good words of length = 4 can be constructed by renaming (permutating) the letters a, b, and c. The input file should be in the same folder as the preparation program is. In the first stage of the loop, there are three output files. One folder was created, which contains two text files for all the so-far-so-good words and all the bad words (see Figure 7). In addition,

the program created one text file which recorded the runtime of the first stage.

In chapter 3.2.1, it was mentioned that the preparation program constructs so-far-so-good words and bad words length by length. It means that, the program uses the result from the last stage of the loop as the input for the next stage (see Figure 8). In this case, the user does not need to copy the input file from the result folder of the last stage. The program can read it from the result folder by checking the length of the words included in the folder's name. Starting from the second stage of the loop, the program prints the runtime of the current stage in the same text file named 'runningTime.txt' which was created at the first stage.

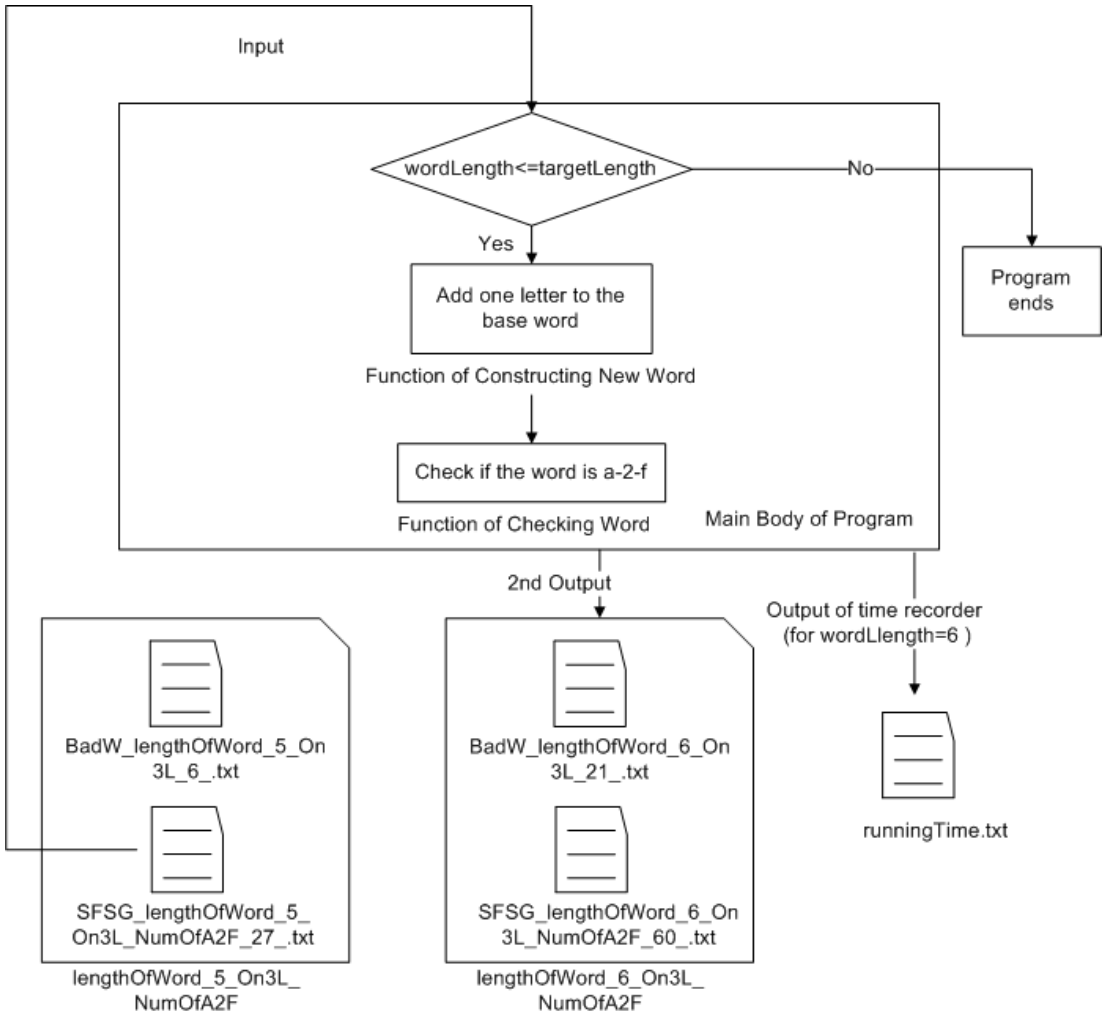


Figure 8. Second Stage of the Loop in Preparation Program

After the final stage finished, the goal of the preparation program was achieved. The text file which contains all the so-far-so-good words in length = 24 was completed (see Figure 9), and then it will be the input for the program which will be introduced in chapter 3.3. Another two text file was accomplished as well. One text file contains all the bad words in length = 24. The other one contains all the records of the runtime for each stage of the loop (see Figure 10). The records of the runtime can help researcher to know how much time the program spent. Indeed, the preparation program can be used to construct the longer lengths of the so-far-so-good words, but this requires a large amount of time.

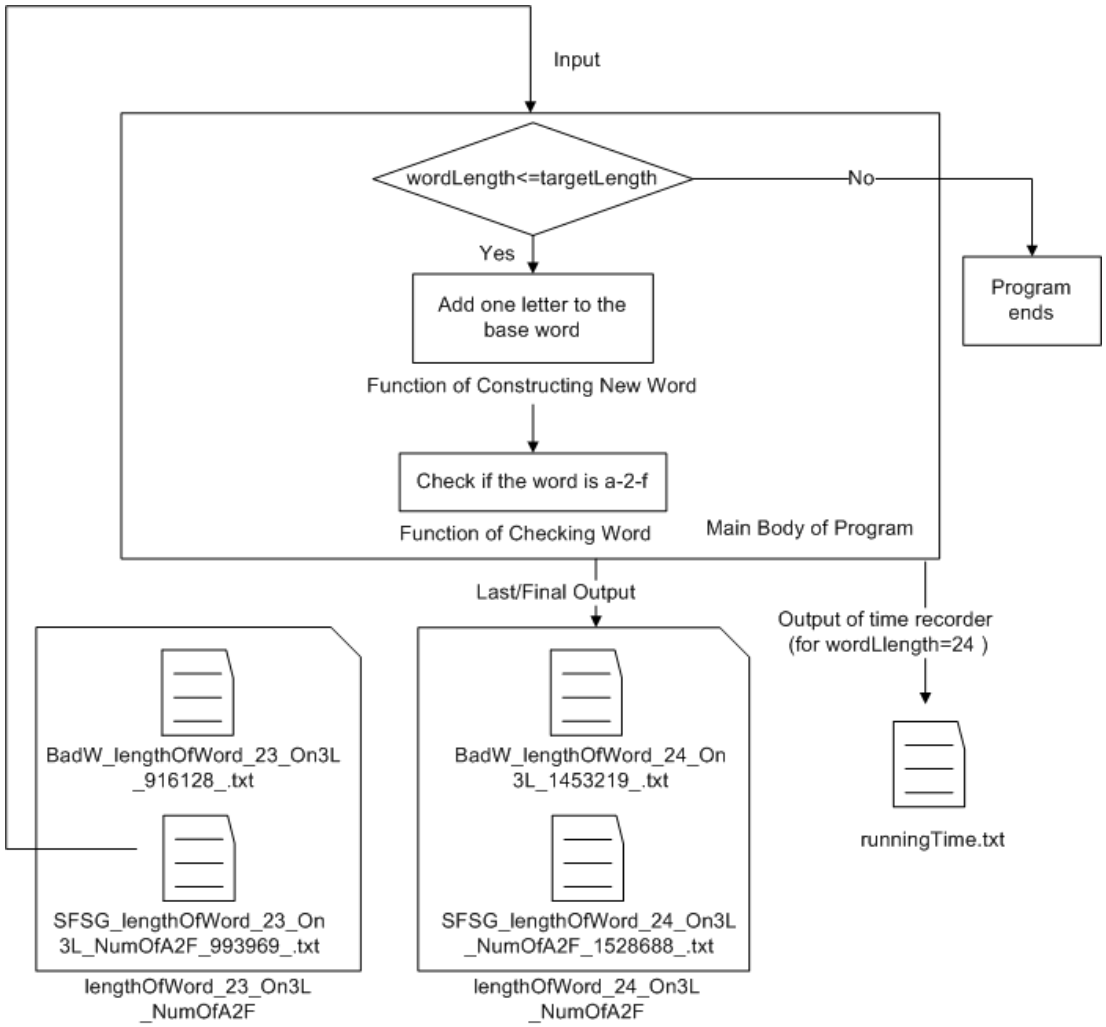
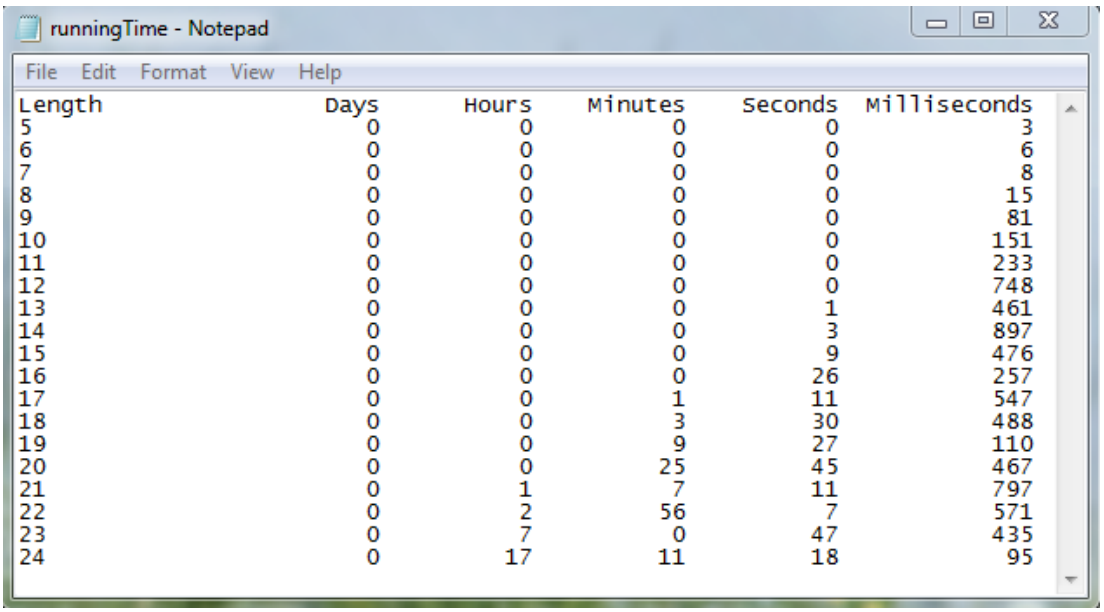


Figure 9. Final Stage of the Loop in Preparation Program



Length	Days	Hours	Minutes	Seconds	Milliseconds
5	0	0	0	0	3
6	0	0	0	0	6
7	0	0	0	0	8
8	0	0	0	0	15
9	0	0	0	0	81
10	0	0	0	0	151
11	0	0	0	0	233
12	0	0	0	0	748
13	0	0	0	1	461
14	0	0	0	3	897
15	0	0	0	9	476
16	0	0	0	26	257
17	0	0	1	11	547
18	0	0	3	30	488
19	0	0	9	27	110
20	0	0	25	45	467
21	0	1	7	11	797
22	0	2	56	7	571
23	0	7	0	47	435
24	0	17	11	18	95

Figure 10. Example of the Running Time Record

3.3 Program for Formal Use

3.3.1 Purpose of Formal Program

The program named ‘asf_3L_FormalUse.py’ (see Appendix 2) is used to construct the longer length of the so-far-so-good words. The formal program directly focuses on constructing the so-far-so-good words in the target length. This method stops at the first so-far-so-good extended word that was found, even if there are still many other so-far-so-good words that could be obtained by extension of the same base word of the input. That is why asf_3L_FormalUse.py is quicker than the preparation program, when only looking for the first so-far-so-good word in each length is focused on.

3.3.2 Overview of Formal Program

In the formal program, there are only two functions in addition to the main function. One function is used to check, if the word is abelian square-free. Another function is used to extend the base word, call the checking function, and return the checking result to the main function.

3.3.3 Explanation of Formal Program

The process of the formal program is slightly different from the preparation program that introduced in chapter 3.2. The main tasks of both programs are

to extend words and to check, if extended words are abelian square-free. However, the methods they use to complete the tasks are different. As it was mentioned in chapters 3.2.1 and 3.2.2, the preparation program does the tasks length by length. The result from the last stage of the loop is the input for the next stage. Instead of length by length, the formal program constructs abelian square-free words directly to the target length for every input word. The example in Figure 11 shows the process of the formal program.

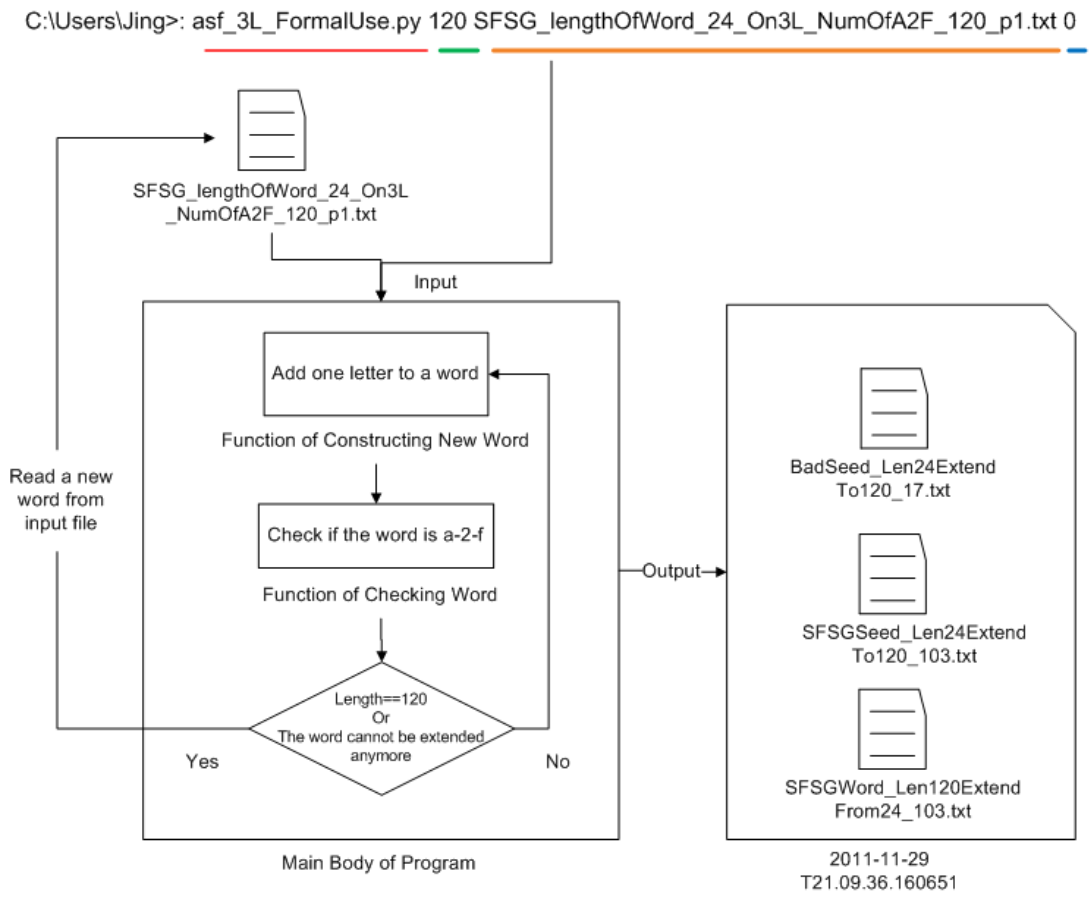


Figure 11. Example of the Process of the Formal Program

The formal program has two types of input. One is the input file which contains the base words and locates in the same folder as the formal program is. Another is the input from the command-line parameters when the users enter a command to run the program. The program supports a simple help on the command line, if the user types the program name which is asf_3L_FormalUse.py, and a space, then types help, the prompting message will be printed on the screen.

On the top of Figure 11 shows an example on how to run the program on the command line. After the program's name (asf_3L_FormalUse.py), the first argument is the target length of the word, which is '120' in the example. The second argument is the name of the text file which contains base words. The third argument is a choice which decides whether print the so-far-so-good words on the target length to a text file. In the example, '0' was assigned the third argument. It means that, the program will not print the so-far-so-good words on the target length to a text file. Otherwise, the program will print the so-far-so-good words of the target length to a text file which is named 'SFSGWord_Len120ExtendFrom24_103.txt' in the example, for any number in addition to zero assigned to the third argument.

The checking function named 'check_abelian_square' achieves the same purpose as the two functions of the preparation program. However, in the formal program, it just combines the two functions for checking starts from the left side and the right side into one function by adding an if-else statement.

Another function named 'find_abelian_square_free' uses the recursion to construct abelian square-free words in target length. The example in Figure 12 shows how the recursion works. The base word is $abac$, its length is four, and the target length is ten.

The most important parts are from step 6 to step 8 in Figure 12. For example, it is supposed that, if the word $caaabacaaa$ in step 8 still contains abelian squares after changed b to c as the prefix of the base, the function will go back to the step 5, and changed a to b as the suffix of the base word. Then the function will check, if extended word is abelian square-free. If not, then the function will repeat to do the same tasks until the word cannot be extended by adding any letter, or the word meets the target length.

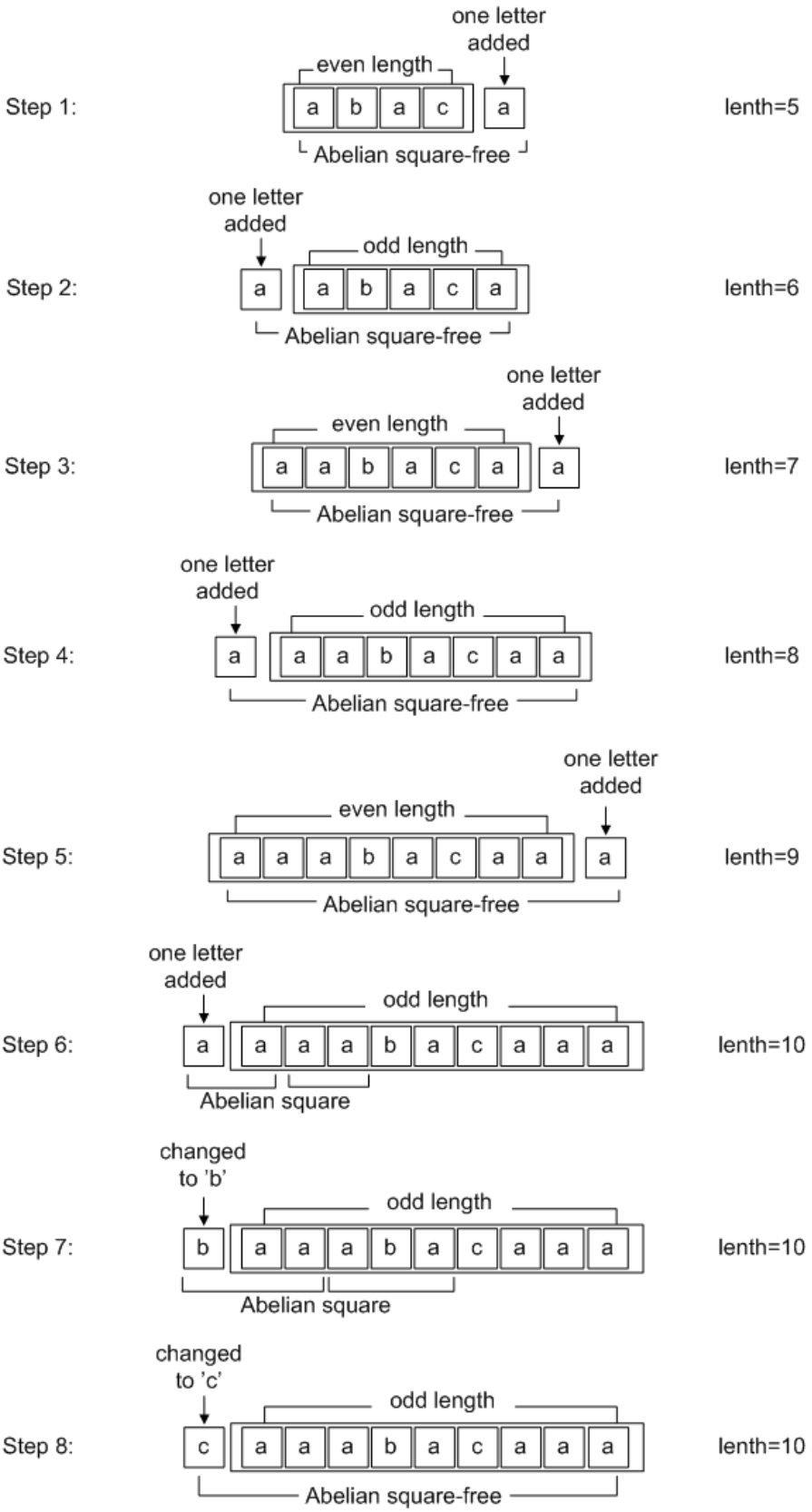


Figure 12. Example of How the Recursion Works in the Function

4 RUNNING ENVIRONMENT ON CSC COMPUTERS

4.1 Background of CSC

The Information Technology Center for Science, i.e., CSC which is a Finnish limited company administrated by the Ministry of Education, Science and Culture. The CSC is a non-profit company which provides IT support only to the research and education fields. There are mainly five services supported by the CSC, data services for science and culture, Funet network services, computing services, application services, and information management services. The customer can apply a set of the services depending on his/her specific requirements. (IT Center for Science Ltd., 2011a.)

The Data service for science and culture concentrates on data storage, data management, data analysis, data processing. It also provides around seventy scientific databases to the related researching. For example, the Finnish academic library systems ARMAS and NELLI are supported by the data service. (IT Center for Science Ltd., 2011b.)

The Funet is a competent and high speed Finnish network which enhances the national or international communication for universities, universities of applied sciences, and research institutions. It supports many modern means for communication, for example, wireless network roaming. Funet is one of the most advanced networks for research and education in the world. (IT Center for Science Ltd., 2011c.)

The computing service provides different resources to meet the customer requirements. The services can be divided into two groups by distance. The first group consists of the local computing servers of the CSC, the more details of which can be found in the next subchapter. The second group is the national and European grid and supercomputing resources. In addition, the related user support and courses also are provided for the customer. (IT Center for Science Ltd., 2011d.)

The application service provides around 200 scientific applications, which allow the customer accesses of the diverse disciplines remotely through the Funet (IT Center for Science Ltd., 2011e). The information management service provides the centralized data management, the maintenance and

development of the information systems, and the reliable consultancy for IT projects. Haka is an example of the maintenance and development of the information system, which is a user identification system of the high level education institutions in Finland. (IT Center for Science Ltd., 2011f.)

There are two groups of the customers in the CSC. Customers from universities and universities of applied sciences can use the CSC services for free. And services for other customers are commercial use. The customer of the CSC can be a person, a group, or an organization. The foreign researcher can apply user account to access the CSC resources. (IT Center for Science Ltd., 2011g.)

4.2 Introduction to CSC Computers

There are four different local computing servers in the CSC namely, Hippu, Louhi, Murska, and Vuori. Hippu is used for interactive and background serial jobs. Vuori and Murska are super clusters which are used for middle size of the parallel jobs. Louhi is the supercomputer which is used for the lager size of the parallel jobs. (IT Center for Science Ltd., 2011h.)

Hippu has a pair of the HP ProLiant DL 785 G5 server which has eight 2.5GHz processors of the quad-core AMD Opteron 8360 SE (Barcelona). Totally, Hippu has 64 cores and 512GB shared memory from both servers. On Hippu, the customer can remotely use the applications which require a big amount of the memory. The program development on Hippu includes Fortran, C, C++, and most common libraries. (IT Center for Science Ltd., 2011i.)

Louhi includes two parts, XT4 and XT5. From both parts, there are 2716 2.3GHz processors of the quad-core AMD Opteron (Barcelona). Totally, Louhi has 10864 cores and around 11.7TB memory. The program development on Louhi includes the C and C++ compilers, the GNU Compiler Collection and the Pathscale Fortran, Portland, and the new Cray Compiling Environment (CCE) Fortran, the C and UPC compilers. The parallel program development includes Message Passing Interface (MPI), AMD Core Math Library (ACML), and LibSci libraries provided by Cray. (IT Center for Science Ltd., 2011l.)

Murska has 1088 2.6GHz processors of dual-core AMD Opteron. Totally Murska has 2176 cores and 5TB memory. The program development on Murska includes the C and C++ compilers, the GNU Compiler Collection and the Pathscale Fortran, Portland. The parallel program development includes the Message Passing Interface (MPI) provided by HP-MPI, (IT Center for Science Ltd., 2011m.)

Vuori has 480 2.6GHz processors of the six-core AMD Opteron. Totally Vuori has 2088 cores and 6.5TB memory. The program development on Vuori is similar to that on Murska. However, on Vuori, there is an OpenMPI library that supports for the MPI (IT Center for Science Ltd., 2011n.)

5 DESCRIPTION OF WORKING WITH CSC COMPUTERS

5.1 Preparation

First, the information about computers on the CSC and the computing services was collected from the CSC website. All the collected information was read, and it helped to make the decision of which services were suitable for the project tasks. Finaly, Hippu and Louhi which are the computing services were chosen. The former is aimed at background serial jobs or interactive jobs, the latter is aimed at the large amounts of the batch jobs. The batch system is a quite new environment, which was not known.

The user accounts for Hippu and Louhi computers were going to be applied. The application forms were filled first. The form includes personal information, contact information, the quota of the CPU time, and the quota of the memory. A short description of the project tasks written by the supervisor of this thesis was enclosed with the application forms.

After the application accepted by the CSC, the user accounts were got and could be used by SSH client programs to log in to Hippu and Louhi, both of which are Unix systems. For example, Nutty SSH and Putty SSH client programs are running on Windows.

5.2 Using Hippu

5.2.1 File Management

For running the program code on Hippu, the program file and the related input file must be uploaded first to the working directory. The scientist's user interface (SUI) is a user friendly environment similar to the Windows user interface (see Figure 13). Folders and files stored on Hippu and Louhi can be managed through the SUI.

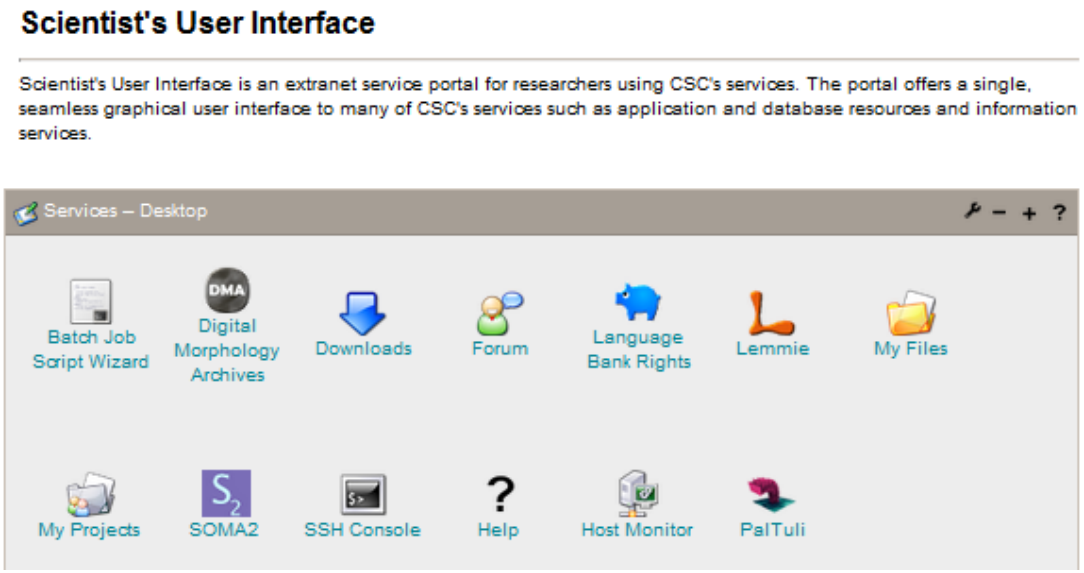


Figure 13. Desktop of Scientist's User Interface (IT Center for Science Ltd., 2011o)

The login page of the SUI can be found from the main page of the CSC. The login system of the SUI is the Haka user identification system introduced in chapter 4.1 (see Figure 14). The user account on the SUI is the same as the one which the customer uses in his/her institution (see Figure 15).

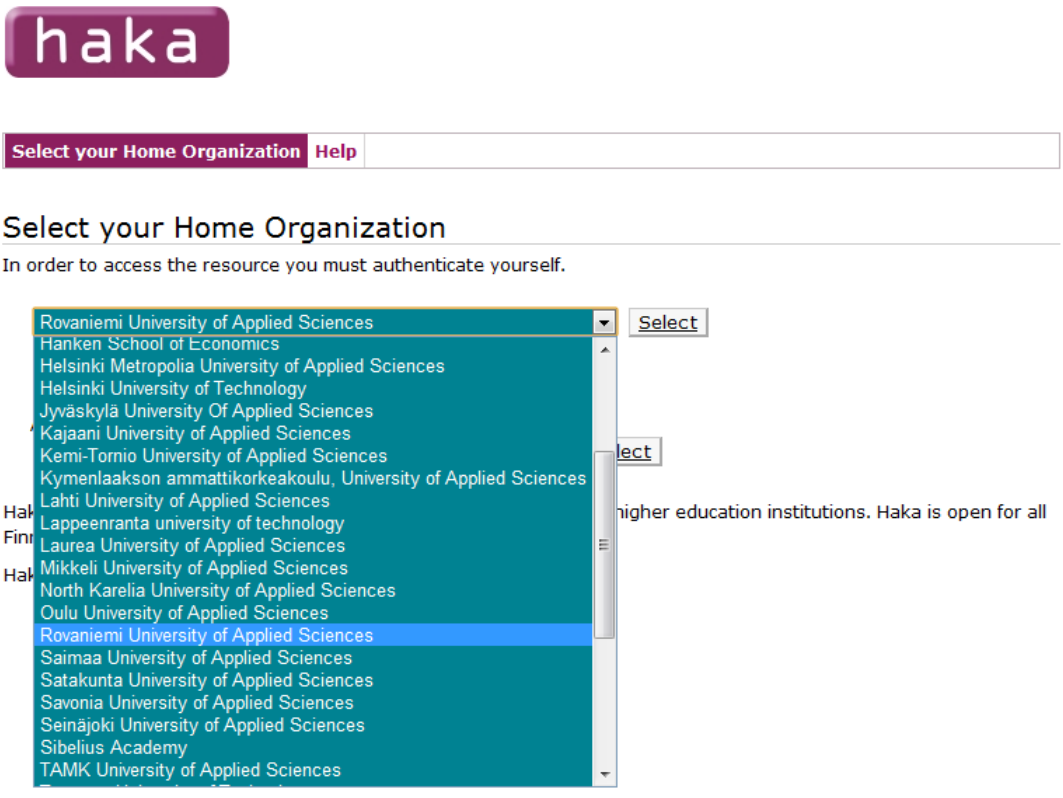


Figure 14. Login Page of SUI_Part 1 (IT Center for Science Ltd., 2011i)

Figure 15. Login Page of SUI_Part 2 (IT Center for Science Ltd., 2011j)

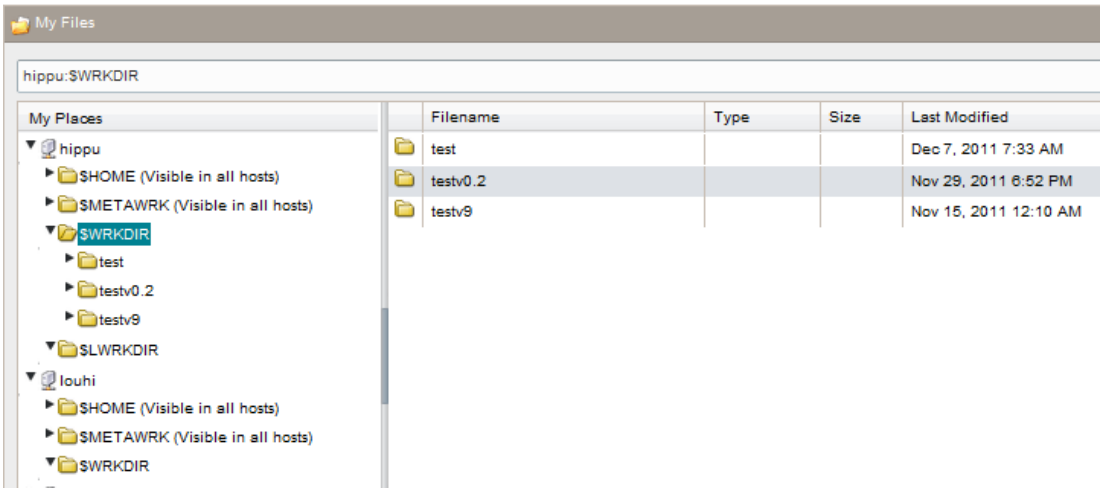


Figure 16. My Files of Scientist's User Interface (IT Center for Science Ltd., 2011p)

In Figure 16, a folder named '$WRKDIR' can be found under the Hippu and Louhi as well. It is the working directory which the program and its input file should be uploaded to. Under the folder named 'My Files' which is shown in Figure 13, the upload function can be found by a right mouse click at the empty place of the interface. Other functions can also be found by a right

mouse click at a file, a folder or the empty place. A folder named ‘test’ was created under the working directory as the place for uploading the programs and the input files.

5.2.2 Running Program

After uploading all the files that are needed for running the program, the Nutty SSH client program can be used to open the secure connection with Hippu. The tool named ‘SSH Console’ which places on the desktop of the SUI has the same function (see Figure 13).

In the window of the configuration, the host name of Hippu which is hippu.csc.fi should be typed in a corresponding place which is marked by a red circle (see Figure 17). The host name can be saved as a session for the next time use. After clicking the open button on the bottom of the window, a Nutty-window (see Figure 18) is opened for the user to log in. Here, the user account is different from what was used to log in the SUI. The user account used here is from the CSC paper which was sent to the customer after the application had been accepted.

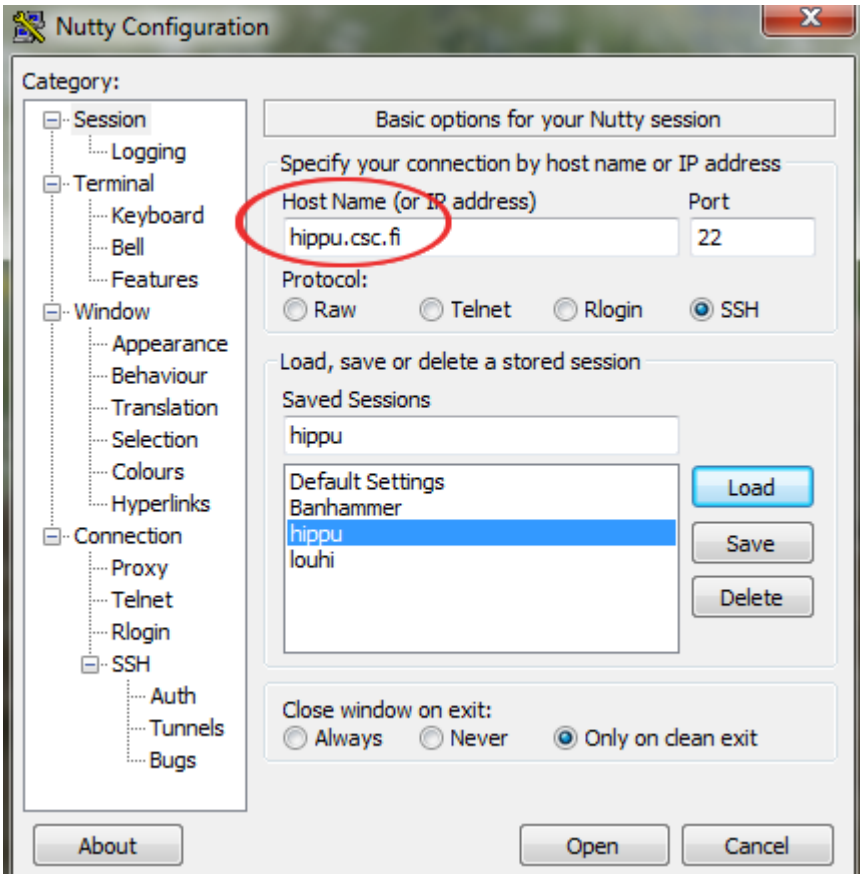


Figure 17. Configuration of Connection with Hippu

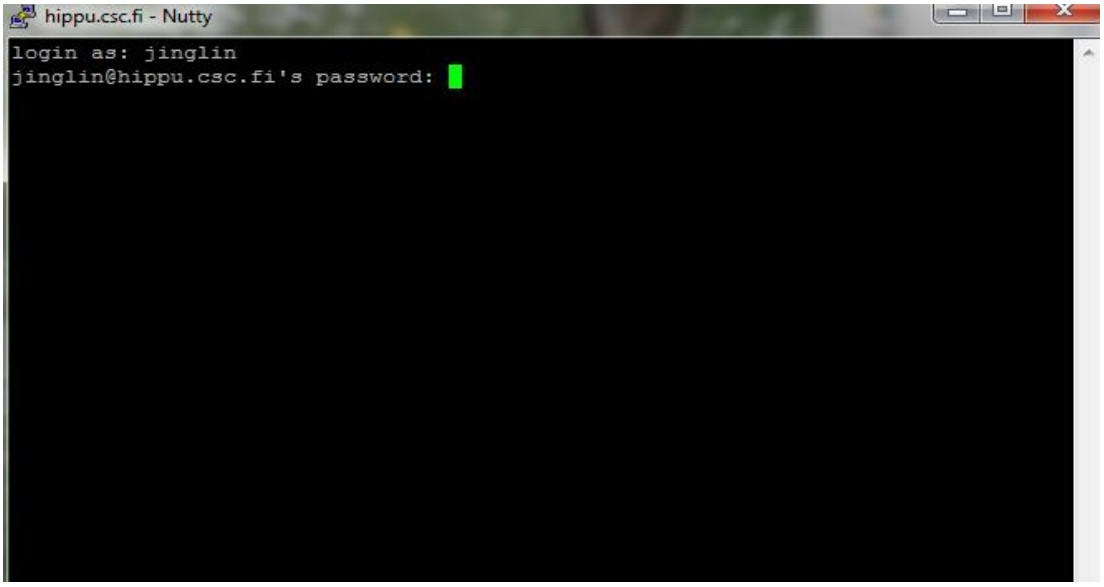


Figure 18. Login Nutty-Window for Hippu

After login, the current path on Hippu is the home directory which can be found in Figure 16, named '$HOME'. Following the steps shows in Figure 19, it can arrive at the path which the program was uploaded. When the current path is same as the program locates, the program can be tested (see Figure 16 and Figure 19).

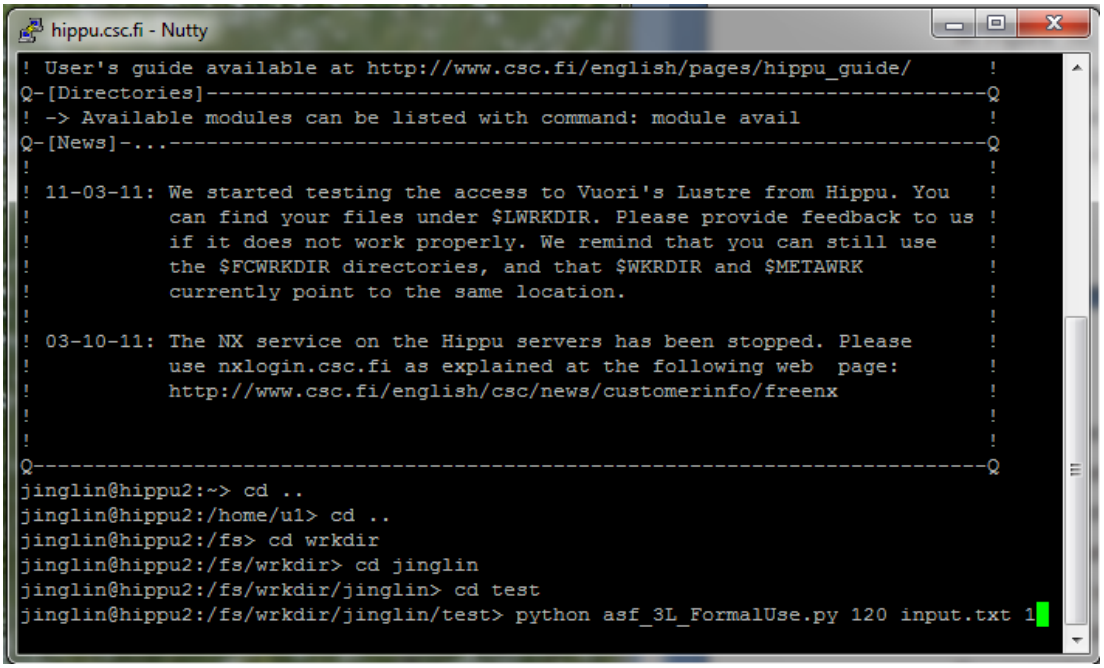


Figure 19. Example of Running the Program in Interactive Way

In Figure 19, the program was run in an interactive way. The command ‘screen’ can be added at the beginning of the last command line to change the way from interactive to the background (see Figure 20).

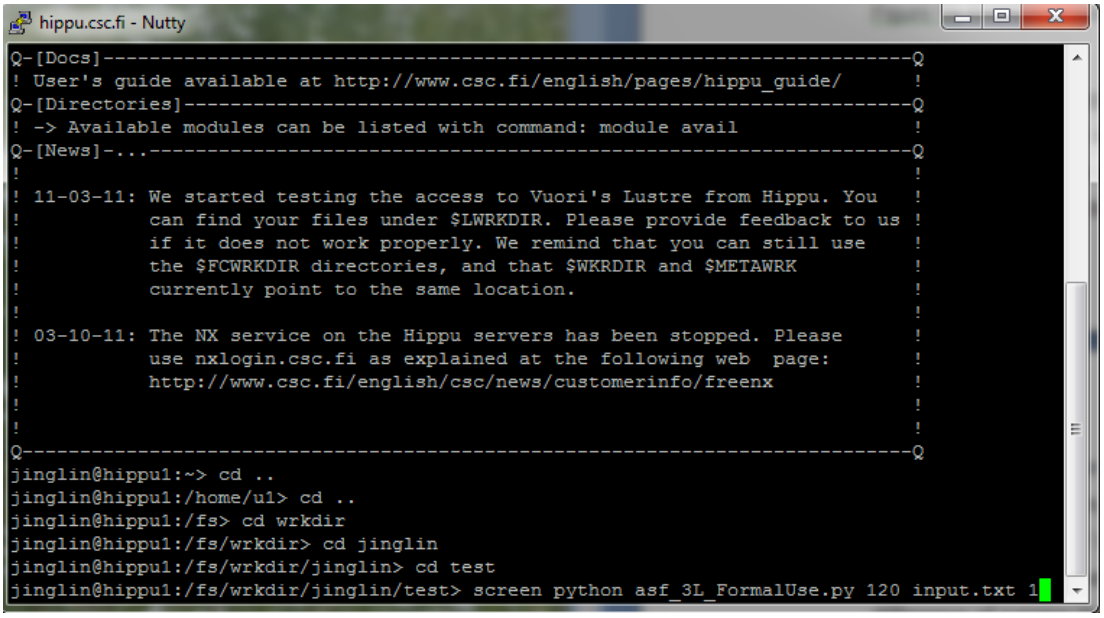


Figure 20. Example of Running the Program in Background Way

After the program has started running in the background, the three keys, Ctrl+A+D, can be used to go back the command line from the background running screen (see Figure 21).

Figure 21. Example of Background Running Screen on Hippu

After came back to the command line, the command ‘logout’ is used to log out off Hippu (see Figure 22). The command ‘screen –rd’ is used to check, whether the program has finished running in the background environment at the next time when logging in to Hippu (see Figure 23). If the screen shows ‘There is no screen to be detached.’ after the command ‘screen -rd’ was entered, it means that, the program had finished running in the background. (see Figure 23).

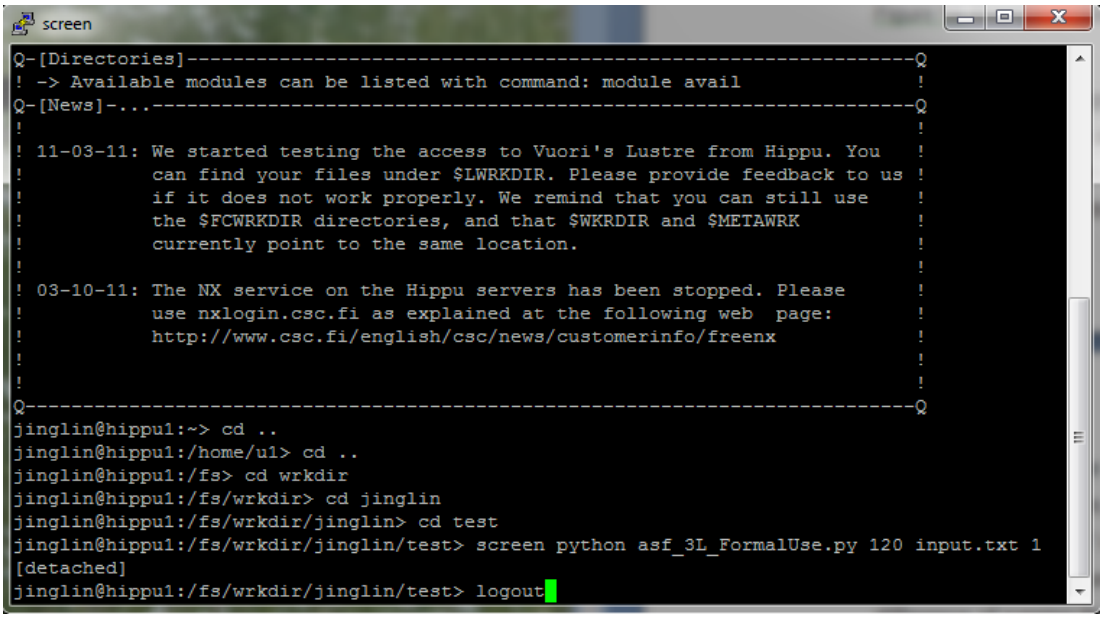


Figure 22. Example of Logout-Screen on Hippu

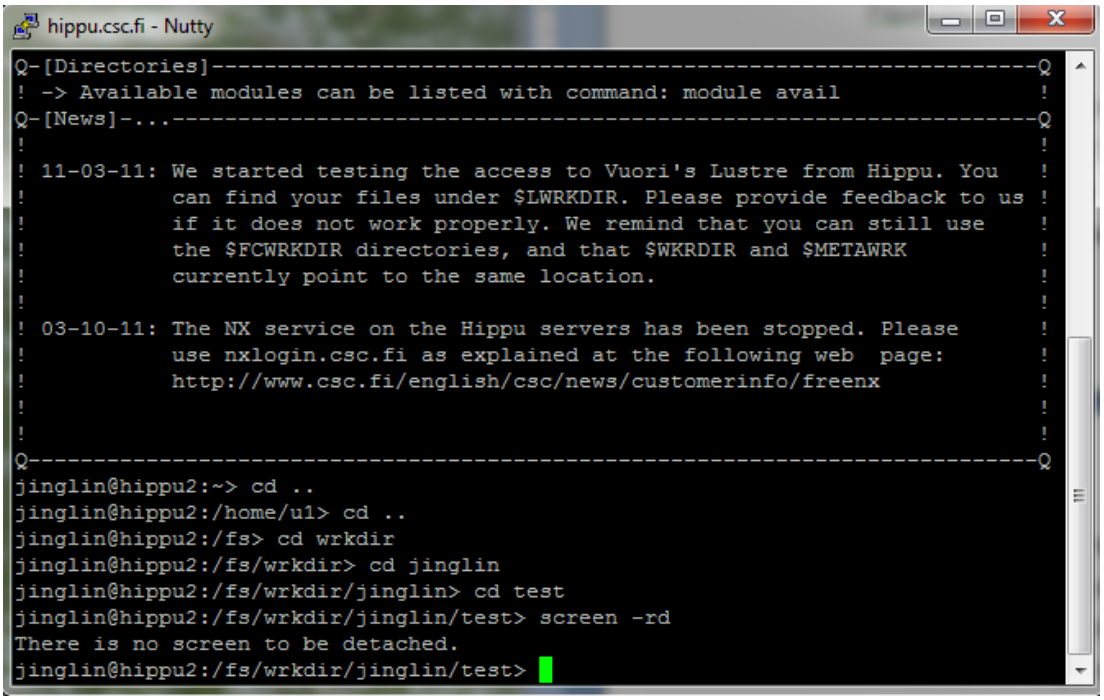


Figure 23. Example of the Command ‘screen –rd’

The difference of running the Python program between Windows and Unix, is the fact that on Unix ‘python’ should be added before the name of the program. For example, for running test.py on the Unix system, ‘python test.py’ should be typed on the command line. But on Windows, only one singly types ‘test.py’ on the command line. To make any Python script executable on Unix, line ‘#!/usr/local/bin/python’ or path to Python interpreter currently being used, must be added to the beginning of the source code. After that, Python script can be run by typing the program’s name also on Unix.

5.3 Description of Results

Table 1 shows the numbers of the favourable and unfavourable words from length = 5 to length = 24 which are from the results by running the preparation program on Hippu.

From the last column which named ‘Percentages of So-far-favourable Words’ in Table 1, it shows the percentage of the so-far-favourable words decreases when the length of the word increases. It will be good news for the further research, if the decrease continues when the length of the word increases. It means that, the amount of the memory which is used to store the list of the so-far-favourable words would not become too big when the words became longer.

It took more than one week for 26279 so-far-favourable words in length = 24 to be extended to length = 120 by the program that was introduced in chapter 3.3. After the program was run, the results show that, there are 20 unfavourable words which cannot be extended to length = 120 and 26259 so-far-favourable words which can be extended to length = 120. But compared to the total number of the so-far-favourable words in length = 24 which is 1528688 in Table 1, the results from these 26279 words cannot be presented as a whole.

The results mentioned above were obtained from the program runs on Hippu. No results were obtained from Louhi because of the problems related to its usage. The parallel program failed to be built in practice due to lack of the proper source material and expertise in parallel programming with Python.

Table 1. Percentages of So-Far-Favourable Words in Each Length

Length of Word	Number of Unfavourable Words	Number of So-Far-Favourable Words	Percentages of So-Far-Favourable Words
5	6	27	81.8%
6	21	60	74.0%
7	49	131	72.8%
8	131	262	66.7%
9	267	519	66.0%
10	582	975	62.6%
11	1080	1845	63.1%
12	2126	3409	61.6%
13	3944	6283	61.4%
14	7558	11291	59.9%
15	13859	20014	59.1%
16	25483	34559	57.6%
17	44892	58785	56.7%
18	78900	97455	55.3%
19	132603	159762	54.6%
20	223512	255774	53.4%
21	360190	407132	53.1%
22	584697	636699	52.1%
23	916128	993969	52.0%
24	1453219	1528688	51.3%

6 CONCLUSION

The problem whether abelian squares (allow xx) are avoidable over three letters has not yet been solved. It requires extremely much work and seems to need assistant tools such as high-performance computers to decrease the runtime of the programs. The results of the 26259 so-far-favourable words which were extended from length = 24 to length = 120, have not yet been proved their correctness.

A good algorithm of the program can help to decrease its runtime and reduce the requirement of the storage memory. For future research, more information about parallel computing and running Python on Louhi should be collected so as to help the research work on a parallel computing environment. In addition, the GRID environment could also be used to help the further research. The running speed of the programs that were run on Hippu could be improved in the future.

BIBLIOGRAPHY

Bean, D.R. – Ehrenfeucht, A. – McNulty, G.F. 1979. Avoidable Patterns in Strings of Symbols. Pacific Journal of Mathematics. 85/2, 261-294.

IT Center for Science Ltd. 2011a. About CSC. Address: http://www.csc.fi/english/csc. Accessed 29 November 2011.

–2011b. About CSC. Data services for science and culture. Address: http://www.csc.fi/english/csc/overview/service_areas/data_storage_services/index_html. Accessed 29 November 2011.

–2011c. About CSC. Funet network services. Address: http://www.csc.fi/english/csc/overview/service_areas/funet. Accessed 29 November 2011.

–2011d. Computing services. Address: http://www.csc.fi/english/research/Computing_services. Accessed 29 November 2011.

–2011e. About CSC. Application services. Address: http://www.csc.fi/english/csc/overview/service_areas/application_services/index_html. Accessed 29 November 2011.

–2011f. About CSC. Information management services. Address: http://www.csc.fi/english/csc/overview/service_areas/administration_services. Accessed 29 November 2011.

–2011g. Welcome as CSC's customer. Address: http://www.csc.fi/english/customers. Accessed 29 November 2011.

–2011h. Computing services. Computing servers. Address: http://www.csc.fi/english/research/Computing_services/computing/index_html. Accessed 29 November 2011.

–2011i. Computing services. Hippu: HP ProLiant DL785 G5 server pair. Address:http://www.csc.fi/english/research/Computing_services/computing/servers/hippu. Accessed 29 November 2011.

–2011j. Select Your Home Organization. Address: https://haka.funet.fi/shibboleth/WAYF?entityID=https%3A%2F%2Fsui.csc.fi%2Fshibboleth&return=https%3A%2F%2Fsui.csc.fi%2FShibboleth.sso%2FDS%3FSAMLDS%3D1%26target%3Dcookie%253Ad6966c12. Accessed 29 November 2011.

–2011k. Kirjautuminen HAKA-federaation palveluun. Address: https://login.ramk.fi/idp/Authn/UserPassword. Accessed 29 November 2011.

–2011l. Computing services. Louhi: Cray XT4/XT5. Address: http://www.csc.fi/english/research/Computing_services/computing/servers/louhi. Accessed 29 November 2011.

–2011m. Computing services. Murska: HP CP4000 BL ProLiant supercluster. Address:http://www.csc.fi/english/research/Computing_services/computing/servers/murska. Accessed 29 November 2011.

–2011n. Computing services.Vuori: HP CP4000 BL ProLiant supercluster. Address:http://www.csc.fi/english/research/Computing_services/computing/servers/vuori. Accessed 29 November 2011.

–2011o. Scientist's User Interface. Address: https://sui.csc.fi/group/sui. Accessed 7 December 2011.

–2011p. Scientist's User Interface. Address: https://sui.csc.fi/group/sui/my-files. Accessed 7 December 2011.

Karhumäki, J. 2011. Combinatorics of Words. Address: http://www.math.utu.fi/en/home/karhumak/combwo.pdf. Accessed 21 October 2011.

Keränen, V. 1986.On the K-Freeness of Morphisms on Free Monoids. AnnalesAcademia ScientiarumFennica Series A. I. Mathematica Dissertationes 61. Helsinki: University of Helsinki.

Keränen, V. 2009a. A Powerful AbelianSquare-Free Substitution over 4 Letters. Theoretical Computer Science. 410/2009, 3893-3900.

–2009b. Combinatorics on Words. The Mathematica Journal. 11/3, 358-375.

Appendices

Appendix 1

preparation_asf3L_v1.0.py

```
#-----------------------------------------------------------------#
# File name: preparation_asf3L_v1.0
# Goal: To generate the lists of all the so-far-so-good words and
#       bad words of the length in the range of input
# Programmer: Jing Lin
#-----------------------------------------------------------------#

import math, sys, os, datetime, locale
locale.setlocale(locale.LC_NUMERIC, "")
#-----------------------------------------------------------------#
# Function format_number is used to format numbers in the output
#   file: runningTime.txt
# Returns the argument as string if number cannot be formatted
#-----------------------------------------------------------------#
def format_number(number):
    try:
        float_number = float(number)
        return locale.format("%.*f", (0, float_number), True)
    except (ValueError, TypeError):
        return str(number)

#-----------------------------------------------------------------#
# Function get_width is used to return widths for printing pretty
#   tables for runningTime.txt
#-----------------------------------------------------------------#
def get_width(column_index):
    try:
        return {
            0:8,   #word length
            1:15,  #days
            2:9,   #hours
            3:9,   #minutes
            4:9,   #seconds
            5:12,  #milliseconds
        }[column_index]
    except:
        return 15

#-----------------------------------------------------------------#
# Function write_table_data is used to print pretty tables to file:
#   runningTime.txt
#-----------------------------------------------------------------#
def write_table_data(target_file, table_data):
    file = open(target_file, 'a+')
```

```
45.     column_paddings = []
46.
47.     for i in range(len(table_data[0])):
48.         column_paddings.append(get_width(i))
49.
50.     for row in table_data:
51.         #for left side only
52.         file.write(format_number(row[0]).ljust
53.                    (column_paddings[0] + 1))
54.         #for other columns
55.         for i in range(1, len(row)):
56.             file.write(format_number(row[i]).rjust
57.                        (column_paddings[i] + 2))
58.         file.write(os.linesep)
59.
60.     file.close()
61.
62. #-------------------------------------------------------------------#
63. # Function createOrOpenTimeF() to create/open the file:
64. #   runningTime.txt
65. #-------------------------------------------------------------------#
66. def createOrOpenTimeF():
67.     try:
68.         times = open('runningTime.txt', 'r+')
69.         times.close()
70.     except:
71.         write_table_data(
72.             'runningTime.txt',
73.             [['Length', 'Days', 'Hours', 'Minutes', 'Seconds',
74.               'Milliseconds']]
75.         )
76.
77. #-------------------------------------------------------------------#
78. #Function checkLoadFileE() checks if loading file exists
79. #-------------------------------------------------------------------#
80. def checkLoadFileE(lastFolderName_results):
81.     fileName_load = ''
82.     for item in os.listdir(lastFolderName_results):
83.         if str(item).endswith('.txt') and \
84.                 str(item).find('BadW') == -1:
85.             fileName_load=str(item)
86.             break
87.     if len(fileName_load) == 0:
88.         print 'The load file is not exist!'
89.         sys.exit()
90.     return fileName_load
91.
92. #-------------------------------------------------------------------#
93. #function checkAndLoadFile checks whether loading file is correct,
      #if it is correct then function will copy its words to a list
94. #-------------------------------------------------------------------#
```

```
95. def checkAndLoadFile(lastFolderName_results, fileName_load, wordLenS):
96.     loadList=[]
97.     startPosition = fileName_load.find('lengthOfWord_')+\
98.             len('lengthOfWord_')
99.     endPosition = fileName_load.find('_On3L')
100.        loadNum = int(fileName_load[startPosition:endPosition])
101.
102.        if loadNum != wordLenS :
103.            print 'The load file is not correct!'+ \
104.                'Please check it!'
105.            sys.exit()
106.
107.        f = open(os.path.join(lastFolderName_results,
108.                fileName_load), 'r')
109.        '''
110.      to copy every line from the file by removing '\n',
111.      '\n\r', etc. from the end of each line
112.      '''
113.       for line in f:
114.           loadList.append(line.rstrip())
115.
116.       f.close()
117.
118.       return loadList[:]
119.
120.   #-----------------------------------------------------------------#
121.   #to extend the word by adding one letter on left side when length
122.   #of base words are odd, and adding one letter on right side when
123.   #length of base words are even
124.   #-----------------------------------------------------------------#
125.   def extendWord(wordLenS, baseWords):
126.       extendedWords=[]
127.       baseLetters = ['a', 'b', 'c'] #3 letters case
128.
129.       if wordLenS%2 != 0:
130.           for bw in baseWords:
131.               for bl in baseLetters:
132.                   extendedWords.append(bl+bw)
133.       else:
134.           for bw in baseWords:
135.               for bl in baseLetters:
136.                   extendedWords.append(bw+bl)
137.       return extendedWords[:]
138.
139.   #-----------------------------------------------------------------#
140.   #function createOrOpenFolder when the folder exist, then just opens
141.   #the folder when the folder did not exist, then create the folder
142.   #-----------------------------------------------------------------#
143.   def createOrOpenFolder(folderN):
144.       try:
145.           os.mkdir(folderN)
```

```
146.        except:
147.            pass
148.
149.    #-----------------------------------------------------------------#
150.    # Function remove_duplicates is used to remove dublicates in a list
151.    # Returns the result as a list
152.    #-----------------------------------------------------------------#
153.    def remove_duplicates(list_arg):
154.        result_list = [] #used to return unique items
155.        for item in list_arg:
156.            #if item is not yet on the result list, add it there
157.            if item not in result_list:
158.                result_list.append(item)
159.        return result_list
160.
161.    #-----------------------------------------------------------------#
162.    # Function a2fCheckL is used to check the words from their left
163.    #   side when the word length is odd
164.    # Returns the bad words list
165.    #-----------------------------------------------------------------#
166.    def a2fCheckL(list_word, int_wordLen):
167.        badWL=[]
168.        max_i = int_wordLen/2
169.        check_for='abc'
170.
171.        for rw in list_word:
172.            for i in range(2,max_i+1):
173.                for j in range(i,int_wordLen+1-i,i):
174.                    part1_check = {'a':0, 'b':0, 'c':0}
175.                    part2_check = {'a':0, 'b':0, 'c':0}
176.
177.                    part1 = rw[j-i : j]
178.                    part2 = rw[j : i+j]
179.
180.                    for char1 in check_for:
181.                        for char2 in part1:
182.                            if char1 == char2:
183.                                part1_check[char1] += 1
184.                        for char3 in part2:
185.                            if char1 == char3:
186.                                part2_check[char1] += 1
187.
188.                    #to put bad words into bad words list
189.                    if part1_check == part2_check:
190.                            badWL.append(rw)
191.                            break
192.
193.        return remove_duplicates(badWL)
194.
195.    #-----------------------------------------------------------------#
196.    # Function a2fCheckR is used to check the words from their
```

```
#   right side when the word length is even
# Returns the bad words list
#-------------------------------------------------------------------#
def a2fCheckR(list_word, int_wordLen):
    badWR=[]
    max_i = (int_wordLen-1)/2
    check_for='abc'

    for rw in list_word:
        for i in range(2,max_i+1):
            for j in range(0,int_wordLen+1-i,i):
                part1_check = {'a':0, 'b':0, 'c':0}
                part2_check = {'a':0, 'b':0, 'c':0}
                x = int_wordLen-j
                if x-2*i < 0:
                    break
                part1 = rw[x-i : x]
                part2 = rw[x-2*i : x-i]

                for char1 in check_for:
                    for char2 in part1:
                        if char1 == char2:
                            part1_check[char1] += 1
                    for char3 in part2:
                        if char1 == char3:
                            part2_check[char1] += 1

                #to put bad words into bad words list
                if part1_check == part2_check:
                    badWR.append(rw)
                    break

    return remove_duplicates(badWR)

#-------------------------------------------------------------------#
# Function removeBadWords is used to remove the bad words away
# Returns the good words list
#-------------------------------------------------------------------#
def removeBadWords(list_OriginalW,list_badWords):
    return list(set(list_OriginalW) - set(list_badWords))

#-------------------------------------------------------------------#
#function writeWordF writes so far so good words or bad words in
#respective files
#-------------------------------------------------------------------#
def writeWordF(folderN, fileN, wordsL):
    f = open(folderN+'/'+fileN+str(len(wordsL))+'_.txt', 'w')

    for word in wordsL:
        f.write(word+ os.linesep)
    f.close()
```

```
247.
248.  #-------------------------------------------------------------------#
249.  #function writeTime writes the running time in the end of every
250.  #word processing file
251.  #---fileResult
252.  #-------------------------------------------------------------------#
253.  def writeTime(wordLen, startT):
254.      # write recorded times
255.      endtime = datetime.datetime.now() - startT
256.      days = endtime.days
257.      hours = endtime.seconds/3600
258.      seconds = endtime.seconds%3600
259.      minutes = seconds/60
260.      seconds = seconds%60
261.      milliseconds = endtime.microseconds/1000
262.
263.      write_table_data(
264.              'runningTime.txt',
265.              [[wordLen, days, hours, minutes,
266.              seconds, milliseconds]]
267.      )
268.
269.
270.  #-------------------------------------------------------------------#
271.  # Main function
272.  #-------------------------------------------------------------------#
273.  def main():
274.
275.      print "Please input the lenght of base word: "
276.      wordLenB = input()
277.      print "Please input the lenght of target word: "
278.      wordLenE = input()
279.
280.      #to check length of object/starting words are correct
281.      if wordLenE <= wordLenB :
282.          print 'The length of object word should be bigger than'+\
283.                ' starting word!'
284.          sys.exit()
285.
286.      #define the variables
287.      wordLenS=wordLenB
288.      createOrOpenTimeF()
289.
290.      while wordLenS < wordLenE:
291.
292.          #define the changing variables
293.          #wordBase=wordLen-1
294.          lastFolderName_results= 'lengthOfWord_'+\
295.              str(wordLenS)+'_On3L_NumOfA2F'
296.          folderName_results= 'lengthOfWord_'+\
297.              str(wordLenS+1)+'_On3L_NumOfA2F'
```

```
298.            fileName_results= 'SFSG_lengthOfWord_'+\
299.                str(wordLenS+1)+'_On3L_NumOfA2F_'
300.            fileName_badWords='BadW_lengthOfWord_'+str(wordLenS+1)+\
301.                              '_On3L_'
302.            resultWords=[] #for putting the a-2-f words
303.            baseWords=[] #for putting the loading words
304.            badWords=[] #for putting the bad words
305.            #variables for recording elapsed time
306.            startTime = datetime.datetime.now()
307.
308.            fileName_load=checkLoadFileE(lastFolderName_results)
309.            baseWords=checkAndLoadFile(lastFolderName_results,
310.                                       fileName_load, wordLenS)
311.            resultWords=extendWord(wordLenS, baseWords)
312.            createOrOpenFolder(folderName_results)
313.
314.            if wordLenS%2 != 0:
315.                badWords = a2fCheckL(resultWords, wordLenS+1)
316.                resultWords = removeBadWords(resultWords,badWords)
317.            else:
318.                badWords = a2fCheckR(resultWords, wordLenS+1)
319.                resultWords = removeBadWords(resultWords,badWords)
320.
321.            #output
322.            writeWordF(folderName_results, fileName_badWords, badWords)
323.            writeWordF(folderName_results, fileName_results,
324.                       resultWords)
325.
326.            #writes the last length of word to the text file
327.            progress = open('progress.txt', 'w')
328.            progress.write(str(wordLenS+1))
329.            progress.close()
330.
331.            writeTime(wordLenS+1, startTime)
332.            wordLenS += 1
333.
334.    # If program is run, run the main function.
335.    if __name__ == "__main__":
336.        main()
```

asf_3L_FormalUse.py

```
1.  import math, os, sys, time
2.  from datetime import datetime
3.
4.  """
5.  Function for checking if a word is abelian square.
6.
7.  Arguments:
8.
9.  word: Word to be checked.
10.
11. length_offset: If zero, the length used to check if word length is
12. even or odd is calculated from given word. Can be changed to be
13. word example -1, then length of the word - 1 will be used to
14. calculate if length is even or odd. Default value 0.
15.
16. Returns True or False
17. """
18. def check_abelian_square(word, length_offset=0):
19.     word_length = len(word)
20.
21.     # Loop through lengths 2 to half of the word's length
22.     # and check if the checked parts of the word are permutations
23.     # of each other.
24.     for i in xrange(2, word_length / 2 + 1):
25.
26.         # Even number, check from right side:
27.         if (word_length - length_offset) % 2 == 0:
28.             right_letters = sorted(list(word[(-i):]))
29.             left_letters = sorted(list(word[(-i * 2):(-i)]))
30.
31.         # Odd number, check from left side:
32.         else:
33.             left_letters = sorted(list(word[(0):(0 + i)]))
34.             right_letters = sorted(list(word[(0 + i):(0 + i * 2)]))
35.
36.         # If ordered lists made from checked parts of the word are
37.         # same, the word is abelua square.
38.         if left_letters == right_letters:
39.             return True
40.
41.     return False
42.
43. """
44. Function for checking if given word is abelian square free when ex-
45. panded to given length. Writes results in a file. Uses recursion.
46.
47. Arguments:
48.
```

```
word: Word to be checked.

base_letters: Words are constructed using these letters.

limit: Target length of the word to be expanded. Default 100.

folder: Folder for result file. Default 'temp'.

Returns True or False
"""
def find_abelian_square_free(word, base_letters, limit=100,
                             folder='temp', write_results_to_file=1):
    word_length = len(word)

    # Loop through base letters and try each letter on word until
    # abelian square free word is found. If not found, return to
    # earlier recursion level. If found and limit is reached,
    # write word to a file and exit the recursion. If found and
    # limit is not reached, continue deeper into recursion.
    for letter in base_letters:

        # Even number, add to right side:
        if word_length % 2 == 0:
            new_word = word + letter

        # Odd number, add to left side:
        else:
            new_word = letter + word

        # Word is not abelian square free, continue loooping
        # through base letters.
        if check_abelian_square(new_word, length_offset=1):
            continue

        # Word is abelian square free and limit has been reached.
        # Write result word into a file and exit the recursion.
        elif len(new_word) == limit:
            if write_results_to_file:
                f = open(os.path.join(folder, 'results.txt'), 'a')
                f.write(new_word + os.linesep)
                f.close()
            return True

        # Word is abelian square free and limit has not been
        # reached. Continue deeper into recursion.
        else:
            word_found = find_abelian_square_free(new_word,
                base_letters, limit, folder,
                write_results_to_file=write_results_to_file)
            if word_found:
                return True

    return False
```

```
102.
103.   """
104.   Main function for this program. Reads system arguments: asf.py
105.   <target word length> <input filename> <write extended words to
106.   file (1 or 0)> or asf.py help. Default values: 120, input.txt,
107.   1.
108.   """
109.   def main():
110.
111.       # Read first argument:
112.       if len(sys.argv) > 1:
113.
114.           if sys.argv[1] == "help":
115.               print "Usage: " + sys.argv[0] + \
116.                     " <TARGET_WORD_LENGTH> <INPUT_FILENAME>" + \
117.                     " <WRITE_EXTENDED_WORDS_TO_A_FILE?>"
118.               sys.exit()
119.
120.           target_length = int(sys.argv[1])
121.       else:
122.           target_length = 120
123.
124.       # Read second argument:
125.       if len(sys.argv) > 2:
126.           input_filename = sys.argv[2]
127.       else:
128.           input_filename = "input.txt"
129.
130.       # Read third argument:
131.       if len(sys.argv) > 3:
132.           write_results_to_file = int(sys.argv[3])
133.       else:
134.           write_results_to_file = 1
135.
136.       # Count good and bad words:
137.       good_word_count = 0
138.       bad_word_count = 0
139.
140.       # Input word length is read from the input file's word's
141.       # lengths.
142.       input_word_length = 0
143.
144.       # Create unique name for the result file folder:
145.       timestamp = datetime.now()
146.       folder_name = timestamp.isoformat().replace(':', '.')
147.       os.mkdir(folder_name)
148.
149.       # Define base letters for constructing words:
150.       base_letters = ['a', 'b', 'c']
151.
152.       # Input file for reading base words:
153.       input_file = open(input_filename, 'r')
154.
```

```
155.        # Loop through all words in input file and find abelian square
156.        # free words for target length. Write results, original bad
157.        # words and original good words in their respective files.
158.        for word in input_file:
159.
160.            # Make sure the word is not an empty string:
161.            if len(word.strip()) > 0:
162.
163.                # If first words, read the input file length from it:
164.                if input_word_length == 0:
165.                    input_word_length = len(word.strip())
166.
167.                # Calculate the time needed for finding abelian
168.                # square free, or for finding out it's not possible
169.                # to find abelian square free word for target length
170.                # for current input file word:
171.                start_time = time.clock()
172.                result = find_abelian_square_free(word.strip(),
173.                    base_letters, limit=target_length,
174.                    folder=folder_name,
175.                    write_results_to_file=write_results_to_file)
176.                end_time = time.clock()
177.
178.                # Abelian square free word found. Write word in file:
179.                if result == 1:
180.                    good_word_count += 1
181.                    good_words_file = open(os.path.join(folder_name,
182.                                                        'good.txt'), 'a')
183.                    good_words_file.write(word.strip() + ' ' + \
184.                                          str(end_time-start_time) + \
185.                                          ' sec' + os.linesep)
186.                    good_words_file.close()
187.
188.                # Abelian square word found. Write word in file:
189.                else:
190.                    bad_word_count += 1
191.                    bad_words_file = open(os.path.join(folder_name,
192.                                                       'bad.txt'), 'a')
193.                    bad_words_file.write(word.strip() + ' ' + \
194.                                         str(end_time-start_time) + \
195.                                         ' sec' + os.linesep)
196.                    bad_words_file.close()
197.
198.        input_file.close()
199.
200.        # If we found abelian square words, rename "bad words file" to
201.        # more descriptive format. Filename has also count of bad
202.        # words.
203.        if bad_word_count > 0:
204.            os.rename(
205.                os.path.join(folder_name, 'bad.txt'),
206.                os.path.join(folder_name, 'BadSeed_Len' + \
207.                    str(input_word_length) + 'ExtendTo' + \
```

```
                    str(target_length) + '_' + \
                    str(bad_word_count) + '.txt')
            )

        # If we found abelian square free words, rename "good words
        # file" to more descriptive format. Filename has also count of
        # good words.
        if good_word_count > 0:
            os.rename(
                os.path.join(folder_name, 'good.txt'),
                os.path.join(folder_name, 'SFSGSeed_Len' + \
                    str(input_word_length) + 'ExtendTo' + \
                    str(target_length) + '_' + \
                    str(good_word_count) + '.txt')
            )
            if write_results_to_file:
                os.rename(
                    os.path.join(folder_name, 'results.txt'),
                    os.path.join(folder_name, 'SFSGWord_Len' + \
                        str(target_length) + 'ExtendFrom' + \
                        str(input_word_length) + '_' + \
                        str(good_word_count) + '.txt')
                )

    # If program is run, run the main function.
    if __name__ == "__main__":
        main()
```